Если вы видите что-то необычное, просто сообщите мне.

Contributing to GHC 3: Hacking Syntax and Parsing

In part 2 of this series, we made more progress in understanding GHC. We got our basic development cycle down and explored the general structure of the code base. We also made the simplest possible change by updating one of the error messages. This week, we'll make some more complex changes to the compiler, showing the ways you can tweak the language. It's unlikely you would make changes like these to fix existing issues. But it'll help us get a better grasp of what's going on. We'll wrap up this series in part 4 by exploring a couple very simple issues.

As always, you can learn more about the basics of Haskell by checking out our other resources. Take a look at our Liftoff Series or download our Beginners Checklist! If you know some Haskell, but aren't ready for the labyrinth of GHC yet, take a look at our Haskell Web Series!

COMMENTS AND CHANGING THE LEXER

Let's get warmed up with a straightforward change. We'll add some new syntax to allow different kinds of comments. First we have to get a little familiar with the Lexer, defined in parser/Lexer.x. Let's try and define it so that we'll be able to use four apostrophes to signify a comment. Here's what this might look like in our code and the error message we'll get if we try to do this right now.

```
module Main where

'''' This is our main function
main :: IO ()
main = putStrLn "Hello World!"
```

```
...

Parser error on `''`
Character literals may not be empty
  |
5 | '''' This is our main function
  | ^^
```

Now, it's easy enough to add a new line describing what to do with this token. We can follow the example in the Lexer file. Here's where GHC defines a normal single line comment:

```
"-- " ~$docsym .* { lineCommentToken }
"--" [^$symbol \ ] . * { lineCommentToken }
```

It needs two cases because of Haddock comments. But we don't need to worry about that. We can specify our symbol on one line like so:

```
"'''''" .* { lineCommentToken }
```

Now we can add the comment above into our code, and it compiles!

ADDING A NEW KEYWORD

Let's now look at how we could add a new keyword to the language. We'll start with a simple substitution. Suppose we want to use the word iffy like we use if. Here's what a code snippet would look like, and what the compiler error we get is at first:

```
main :: IO ()
main = do
  i <- read <$> getLine
  iffy i `mod` 2 == 0
    then putStrLn "Hello"
    else putStrLn "World"

  ...
```

```
Main.hs:11:5: error: parse error on input 'then'
   |
11 |    then putStrLn "Hello"
   |    ^^^^
```

Let's do a quick search for where the keyword "if" already exists in the parser section. We'll find two spots. The first is a list of all the reserved words in the language. We can update this by adding our new keyword to the list. We'll look for the reservedIds set in basicTypes/Lexeme.hs, and we can add it:

```
reservedIds :: Set.Set String
reservedIds = Set.fromList [ ...
  , "_", "iffy" ]
```

Now we also have to parse it so that it maps against a particular token. We can see a line in Lexer.x where this happens:

```
( "if", ITif, 0)
```

We can add another line right below it, matching it to the same ITif token:

```
( "iffy", ITif, 0)
```

Now the lexer matches it against the same token once we start putting the language together. Now our code compiles and produces the expected result!

```
lghc Main.hs
./prog.exe
5
World
```

REVERSING IF

Now let's add a little twist to this process. We'll add another "if" keyword and call it reverseif. This will change the ordering of the if-statement. So when the boolean is false, our code will execute the first branch instead of the second. We'll need to work a little further upstream. We want to re-use

as much of the existing machinery as possible and just reverse our two expressions at the right moment. Let's use the same code as above, except with the reverse keyword. Then if we input 5 we should get Hello instead of World.

```
main :: IO ()
main = do
  i <- read <$> getLine
  reverseif i `mod` 2 == 0
    then putStrLn "Hello"
    else putStrLn "World"
```

So we'll have to start by adding a new constructor to our Token type, under the current if token in the lexer.

```
data Token =
  ...
  | ITif
  | ITreverseif
  ...
```

Now we'll have to add a line to convert our keyword into this kind of token.

```
...
("if", ITif, 0),
("reverseif", ITreverseif, 0),
...
```

As before, we'll also add it to our list of keywords:

```
reservedIds :: Set.Set String
reservedIds = Set.fromList [ ...
  , "_", "iffy", "reverseif" ]
```

Let's take a look now at the different places where we use the ITif constructor. Then we can apply them to ITreverseif as well. We can find two more instances in Lexer.x. First, there's the function maybe_layout, which dictates if a syntactic construct might need an open brace. Then there's the isALRopen function, which tells us if we can start some kind of other indentation. In both of these, we'll follow the example of ITif:

```
maybe_layout :: Token -> P ()
...
  where
    f ITif = pushLexState layout_if
    f ITreverseif = pushLexState layout_if


...
isALRopen ITif = True
isALRopen ITreverseif = True
...
```

There's also a bit in Parser.y where we'll need to parse our new token:

```
%token
 ...
 'if' { L _ ITif }
 'reverseif' { L _ ITreverseif }
```

Now we need to figure out how these tokens create syntactic constructs. This also seems to occur in Parser.y. We can look, for instance, at the section that constructs basic if statements:

```
| 'if' exp optSemi 'then' exp optSemi 'else' exp
  {% checkDoAndIfThenElse $2 (snd $3) $5 (snd $6) $8 >>
   Ams (sLL $1 $> $ mkHsIf $2 $5 $8)
     (mj AnnIf $1:mj AnnThen $4
       :mj AnnElse $7
       :(map (\l -> mj AnnSemi l) (fst $3))
      ++(map (\l -> mj AnnSemi l) (fst $6))) }
```

There's a lot going on here, and we're not going to try to understand it all right now! But there are only two things we'll need to change to make a new rule for reverseif. First, we'll obviously need to use that token instead of if on the first line.

Second, see that mkHsIf statement on the third line? This is where we make the actual Haskell "If" expression in our syntax tree. The $5 refers to the second instance of exp in the token list, and the $8 refers to the third and final expression. These are, respectively, the True and False branch expressions of our "If" statement. Thus, to reverse our "If", all we need to do is flip this arguments on the third line!

```
| 'reverseif' exp optSemi 'then' exp optSemi 'else' exp
  {% checkDoAndIfThenElse $2 (snd $3) $5 (snd $6) $8 >>
   Ams (sLL $1 $> $ mkHsIf $2 $8 $5)
     (mj AnnIf $1:mj AnnThen $4
      :mj AnnElse $7
      :(map (\l -> mj AnnSemi l) (fst $3))
     ++(map (\l -> mj AnnSemi l) (fst $6))) }
```

Finally, there's one more change we need to make. Adding this line will introduce a couple new shift/reduce conflicts into our grammar. There are already 233, so we're not going to worry too much about that right now. All we need to do is change the count on the assertion for the number of conflicts:

```
%expect 235 -- shift/reduce conflicts
```

Now when we compile and run our simple program, we'll indeed see that it works as expected!

```
lghc Main.hs
./prog.exe
5
Hello
```

CONCLUSION

In this part, we saw some more complicated changes to GHC that have tangible effects. In the fourth and final part of this series, we'll wrap up our discussion of GHC by looking at some real issues and contributing via Github.

To learn more about Haskell, you should check out some of our basic materials! If you're a beginner to the language, read our Liftoff Series. It'll teach you how to use Haskell from the ground up. You can also take a look at our Haskell Web Series to see some more advanced and practical skills!

Revision #1
Created 11 March 2022 16:36:20 by gasick
Updated 11 March 2022 17:11:16 by gasick